EXHIBIT
A

Notification Service API Reference

This chapter contains the following topics:

Introduction

The BEA Notification Service supports two application programming interfaces. One is based on the CORBA-based Notification Service as defined by the *CORBAservices: Common Object Services Specification*. This interface is referred to in this document as the CosNotification Service interface. The other interface, called the BEA Simple Events interface, is a BEA proprietary interface designed as an easier to use alternative.

Both interfaces pass structured events as defined by the CORBA-based Notification Service specification and are compatible with each other; that is, events posted using the CosNotification Service interface can be subscribed to by the BEA Simple Events interface and vice versa.

Before using the Notification Service APIs, consider the following topics:

Quality of Service

To determine the persistence of the subscription and whether or not events delivery is retried following a failed delivery, subscribers specify a Quality of Service (QoS). There are two Quality Of Service (QoS) settings: *persistent* and *transient*. The QoS is a property of the subscription.

Persistent Subscriptions

Persistent subscriptions provide strong guarantees about event delivery and the permanence of the subscription. Persistent subscriptions do come with a cost, however, as they consume more system resources (for example, disk space, CPU cycles, and so on), and require more administration (such as managing queues and detecting dead subscribers)

Persistent subscriptions exhibit the following properties:

- The subscription is in effect until an unsubscribe operation is performed. This means that a subscriber application can be shut down and its subscription can still be active. In this case, events are stored for the subscriber and, when the subscriber restarts, are delivered to the subscriber without it having to recreate the subscription.
- If an event cannot be delivered, event delivery is retried until the administrative retry limit is exceeded.When the event

retry limit has been exceeded, the event is moved from the pending queue to an error queue. An administrator can move events from the error queue back to the pending queue, where delivery attempts will restart.

- If an event is successfully delivered to a subscriber, but the Notification Service for some reason does not receive the "successful delivery" return message, the Notification Service may deliver the same event more than once.

Transient Subscriptions

Transient subscriptions provide the best performance with the least overhead.

Transient subscriptions exhibit the following properties:

- One attempt is made to deliver the event to each matching subscription. If that attempt fails, the event is lost.

The subscription is in effect until a failed event delivery is detected. On detection of a failed delivery, the subscription is terminated. Normally, the Notification Service, for performance reasons, does not check whether it successfully delivered an event to a transient subscriber. However, occasionally, when the Notification Service delivers an event to a transient subscriber, it checks whether or not the event was successfully delivered. If it was not successfully delivered and the CORBA::TRANSIENT exception is not returned, the Notification Service assumes that the subscription has gone away and cancels the subscription. If the Notification Service receives the CORBA::TRANSIENT exception when an attempt to deliver fails, it assumes that the subscriber is busy and discards the event, but it does not cancel the subscription.

The automatic cancellation of dead transient subscriptions provides a cleanup mechanism for transient subscribers that forget to unsubscribe. Note, however, that the Notification Service checks for successful deliver the first time it sends an event to a subscriber, but does not perform it again until five minutes have elapsed and it delivers another event. Therefore, the interval between checks is at least five minutes, but will be longer if there is no event to deliver when five minutes have elapsed. The minimum interval of five minutes is fixed and cannot be changed. Therefore, event delivery failure is not necessarily detected on the first failed delivery attempt. It is only detected when the Notification Service checks.

Obtaining the Channel Factory

The Channel Factory is used by event poster applications and subscriber applications to find the event channel. The event channel is then used to post events and to subscribe, or create subscriptions, and unsubscribe, or cancel subscriptions.

Notification Service applications use the Bootstrap object to obtain an object reference to the event channel factory. This is done by using the Tobj_Bootstrap::resolve_initial_references operation. The Bootstrap Object supports two service IDs for Notification Service applications, NotificationService and Tobj_SimpleEventsService . The NotificationService object is used in applications that use the CosNotification Service API. The Tobj_SimpleEventsService object is used in applications that use the BEA SimpleEvents API.

Service ID	Object Type
NotificationService	CosNotifyChannelAdmin::EventChannelFactory
Tobj_SimpleEventsService	Tobj_SimpleEvents::ChannelFactory

Using Transactions

The behavior regarding transactions is the same for the BEA SimpleEvents API and the CosNotification Service API. The only operation that supports transactional behavior is push_structured_event , which is supported by the CosNotifyChannelAdmin::StructuredProxyPushConsumer and Tobj_SimpleEvents::Channel interfaces. All other operations can be used in the context of a transaction, but work the same regardless of whether they are executed in a transaction or not.

The behavior when posting an event is tied to the QoS of the subscription. If an event is posted in the context of a transaction, and the event delivery QoS of the subscription is persistent, the delivery will be affected by the outcome of the transaction; that is, if the transaction is committed,

the Notification Service attempts to deliver the event to subscribers as it normally would. If the transaction is rolled back, then the Notification Service does not attempt to deliver the event.

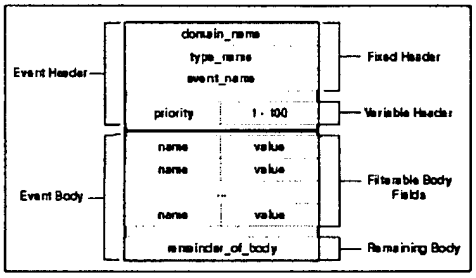
If an event is posted in the context of a transaction, and the event delivery QoS of the subscriber's subscription is transient, one attempt will be made to deliver the event, regardless of the transaction outcome. That is, the transaction has no effect on whether the event is delivered or not, and one attempt will be made to deliver the event.

Note: There is no transaction context associated with event delivery. However, in the case of persistent subscriptions, once the poster's transaction commits, the Notification Service guarantees that the event will be delivered to the subscriber or put on the error queue to await administrative action.

Structured Event Fields, Types and Filters

All events that are either pushed by posters to the Notification Service, or delivered to subscribers, are COS Structured Events, that is, they conform to the definition of Structured Events as specified by the CORBA-based Notification Service-a service which extends the CORBAservices Event Service. If the events are to be filtered based on content (versus filtering on domain and type), or if the events are going to be subscribed to by BEA TUXEDO applications, then additional restrictions apply. The restrictions apply to data types and filtering based on event content. These restrictions are explained on a section-by-section basis below.

Figure 2-1 Structured Event



- The Fixed Header section consists of three fields that can be used when you create structured events: fixed_header.event_type.domain_name and fixed_header.event_type.type_name, and fixed_header.event_type.event_name. When an event is post all three of the these fields are passed in the Notification Service. However, when subscriptions are created, only the first two fields, domain_name and type_name, are used to filter events. These fields are defined in the subscription as regular expressions. The event_name field cannot be used in subscriptions.
- The Variable Header consists of a single name/value (NV) pair, namely Priority. Priority can take a value in the range 1-100 (versus a range of -32767 to 32,767 as specified in CORBA Notification Service specification). Priority is used internally to the system to prioritize the processing of events. The highest priority is 100. There is no guarantee that higher priority events will, in fact, be given priority over lower priority events. The support provided for the Variable Header differs from that specified in CORBA Notification Service specification in two ways: first, there is a single field supported (Priority) versus the five fields listed in the specification; and second, user-defined fields are supported, but no action is taken in response to their content. The user defined fields are merely passed through.
- The Filterable Body consists of zero or more NV pairs. The values in these pairs are limited to the following types: any , long , unsigned long, short , unsigned short, octet, char , float , double , string , boolean , void , and null . These fields can be used in filter expressions.
- The Remaining Body consists of a single ANY . The value is limited to the following types: any , long , unsigned long, short , unsigned short, octet, char , float , double , string , boolean , void , and null .This field cannot be used in a filter expression.

Designing Events

The design of events is basic to any notification service. The design impacts not only the volume of information that is delivered to matching subscriptions, but the efficiency and performance of the Notification Service as well. Therefore, careful planning should be done to ensure that your Notification Service will be able to handle your needs now and allow for future growth.

The Notification Service supports five levels of event design: 1) domain name, 2) type name, 3) priority, 4) filterable data, and 5) remainder of body. When designing an event, you must specify

a domain name and a type name; priority and filterable data are optional. The domain name you choose can relate to your business. Hospitals, for example, are in the health-care business, so for a Notification Service application for a hospital you may choose "HEALTHCARE" as a domain name. You may want to categorize the events by the type of insurance provider, so you may choose "HMO" or "UNINSURED" as the type name. You may want to further define the events by the entity responsible for payment, so you may choose to use the filterable data to identify the entity as "billing" for a specific "HMO_Account" or a specific or "Patient_Account." Listing 2-1 shows an example of this type of event design.

Listing 2-1 Event Design

```
domain_name = "HEALTHCARE"
type_name = "HMO"
#Filterable data name/value pairs.
filterable_data.name = "billing"
filterable_data.value = 4498
filterable_data.name = "patient_account"
filterable_data.value = 37621
```

Obviously, the more specific and precise you are in designing the events that you want your Notification Service application to post and receive, the fewer will be the events the Notification Service will have to process. This has a direct impact system resources and configuration requirements. Therefore, a lot of thought should be given to event design.

Creating FML Field Table Files for Events

You must create Field Manipulation Language (FML) field table files for events only if one of the following capabilities is required; otherwise FML tables are not required.

- Event data filtering (in addition to domain and type fields) between WLE event posters and subscribers
- Interoperability between the WLE Notification Service and the BEA TUXEDO Event Broker

A structured event's filterable_data field contains a list of name/value (NV) pairs. An event's data is typically stored in this list. The field names in the FML field table files must match the name in the structured event. The field type can be any allowable FML type (long , short , double , float , char , string) except carray). The value in the structured event must be the same type as defined in the field table. Table 2-1 shows the CORBA Any types supported by WLE and which of them can be used for data filtering and BEA TUXEDO interoperability.

Table 2-1 Supported CORBA Any Types

CORBA Any Types	Supported for Data Filtering and TUXEDO Interoperability
short	Yes
long	Yes
unsigned short	No
unsigned long	No
float	Yes
double	Yes
char	Yes
boolean	No
octet	No

string	Yes
void	No
null	No
any	No

Listing 2-2 shows an example of an FML field table file. The *base 2000 is the base number for the fields. The first entry has a field name of billing, a field number of 1 relative to the base, and a field type of long.

Listing 2-2 Data Filtering FML Field Table File

```
*base 2000

#Field Name      Field #  Field Type   Flags    Comments
#-----------     -------  ----------   ------   --------
billing          1        long         -        -
stock_name       2        string       -        -
price_per_share  3        double       -        -
number_of_shares 5        long         -        -
```

The following guidelines and restrictions apply to WLE FML field table files:

- The FML file name cannot exceed 15 characters in length.
- Because WLE uses FML32, the base number plus the field number is restricted to be between 101 and 33,554,431, inclusive.
- When FML is used with other software that also uses fields, additional restrictions may be imposed on field numbers.

For information on how to create and configure FML field table files, see field_tables (5) in the *BEA TUXEDO Reference* and the *BEA TUXEDO FML Programmer's Guide*.

Interoperability with BEA TUXEDO Applications

Applications that use the WLE Notification Service are interoperable with BEA TUXEDO applications that use the BEA TUXEDO Event Broker. An application using the WLE Notification Service can post events that are delivered to BEA TUXEDO Event Broker subscribers, and can receive events that have been posted by BEA TUXEDO Event Broker.

To achieve this interoperability, it is necessary understand the mapping between CosNotification Structured Events and BEA TUXEDO FML buffer so that the contents of the FML Field Tables can be coordinated between BEA TUXEDO and WLE. There are two cases to consider: posting events that are to be received by BEA TUXEDO applications via BEA TUXEDO Event Broker; and receiving events that have been posted to the Notification Service Event Channel by BEA TUXEDO applications.

Posting Events

For a BEA TUXEDO application to subscribe to events posted by a WLE application, you must understand how a WLE structured event is mapped to FML32 and the event name at posting time. The mapping is as follows:

- The domain_name and type_name are assembled into a string in the form domain_name.type_name to form the event name. This is the event name (eventname parameter) used on the tppost operation.
- Each name/value (NV) pair in the Filterable Body and the variable header portion of the structured event is mapped to an FML32 field of the same name if the field is also defined in FML. If you set the domain to "TMEVT", then the event name equals the type name.

Receiving Events

BEA TUXEDO System events and User events can be received by WLE applications. System events

are generated by the BEA TUXEDO system-not by applications. User events are generated by BEA TUXEDO applications. For a listing of System events see EVENTS (5) in *BEA TUXEDO Reference*. System events and User events are mapped in CosNotification Structured Events as follows.

Structured Event Fields	Value
domain_name	Always set to "TMEVT"
type_name	Empty string
event_name	Empty string
Variable Header (Priority)	Empty sequence
Filterable Body Fields	Same as FML field name **Note:** Filterable body fields consist of name/value pair, where the name portion is the same as the FML field name.
Remainder of Body	Always set to void

The BEA TUXEDO system detects and posts certain pre-defined events related to system warnings and failures. For example, system-generated events report on configuration changes, state changes, connection failures, and machine partitioning.

In order for a WLE application to receive events posted by a BEA TUXEDO application it is necessary to understand how a FML buffer containing a BEA TUXEDO event is used to fabricate a WLE structured event. It is also necessary to know how the domain_name and type_name are related to the BEA TUXEDO event name. There are two cases to consider, system events and user events.

Note that BEA TUXEDO uses a leading dot (".") in the event name to distinguish system-generated events from application-defined events. An example of a system event is .SysNetworkDropped . An example of a user event is eventsdropped . To subscribe to these events, the Notification Service subscriber application must define the subscription as follows:

- System Event

  domain_name ="TMEVT"
  type_name=".SysNetworkDropped"

- User Event

  domain_name ="TMEVT"
  type_name="eventsdropped"

  When the events are received, the Notification Service subscriber application parses each event as follows:

  domain_name="TMEVT"
  type_name=""
  event_name=""
  variable_header=empty
  Filterable_data=(content of the FML buffer)

Parameters Used When Creating Subscriptions

When you create subscriptions, you can specify the following parameters. These parameters are supported the BEA Simple Events API and the CosNotification Service API.

subscription_name

Specifies a name that identifies the subscription to the Notification Service and the subscriber. Applications should use

names that are meaningful to a system administrator since this is the primary way that an administrator associates an application with a subscription and the events that are delivered to the subscriber via the subscription. This parameter is optional (that is, an empty string can be passed in). More than one subscription can use the same name.

The subscription_name must not exceed 128 characters in length.

domain_type

Same parameter as the domain_type field in the Fixed Header portion of a structured event, as defined by the CORBA-based Notification Service specification. This field is a string that is be used to identify a particular vertical industry domain in which the event type is defined, for example, "Telecommunications", "Finance", and "Health Care". Because this parameter is a regular expression, you can also use it to set domain patterns on which to filter. For example, to subscribe to all domains that begin with the letter F, set the domain to "F.*" . For information about how to construct regular expressions, see the recomp (3) command in *BEA TUXEDO Reference*.

type_name

Same parameter as the type_name field in the Fixed Header portion of a structure event, as defined in the CORBA-based Notification Service specification. It is a string that categorizes the type of event, uniquely within the domain, for example, Comm_alarm, StockQuote, and VitalSigns. Because this parameter is a regular expression, you can also use it to set event type patterns on which to filter. For example, to subscribe to all event types that begin with the letter F, you would set the type to "F.*" . For information about how to construct regular expressions, see the recomp (3) command in *BEA TUXEDO Reference*.

data_filter

Specifies the values of the fields of filterable data and variable header on which you want to filter. For example, a subscription to news stories may have a domain of "News", a type of "Sports", and a data_filter of "Scores > 20".

This parameter defines the data that the subscription must match in Boolean expressions. The following data types are supported: short , long , char , float , double , and string . Table 2-2 lists the Boolean expression operators supported.

Table 2-2 Boolean Expression Operators

Expression	Operators
unary	+, -, !, ~
multiplicative	*, /, %
additive	+, -
relational	< , >, <=, >=, ==, !=
equality and matching	==, !=, %%, !%
exclusive OR	^
logical AND	&&
logical OR	||

To use data filtering, you must setup an FML table, include filters in the subscription filter the data, and post the event. Listing 2-3 shows an example of these tasks.

Listing 2-3 Data Filtering Requirements

```
//Setting up the FML Table

Field table file.
----------
*base 2000

*Field Name    Field #  Field Type  Flags   Comments
-----------    -------  ----------  ------  ------
StockName      1        string      -       -
PricePerShare  2        double      -       -
CustomerId     3        long        -       -
CustomerName   4        string      -       -
```

```
//Subscription data filtering.
1) "NumberOfShares > 100 && NumberOfShares < 1000"
2) "CustomerId == 3241234"
3) "PricePerShare > 125.00"
4) "StockName == 'BEAS'"
5) "CustomerName %% '.*Jones.*'" // CustomerName contains "Jones"
6) "StockName == 'BEAS' && PricePerShare > 150.00"

//Posting the event.
// C++
CosNotification::StructuredEvent ev;
...
ev.filterable_data[0].name    = CORBA::string_dup("StockName");
ev.filterable_data[0].value <<= "BEAS";
ev.filterable_data[1].name    = CORBA::string_dup("PricePerShare");
ev.filterable_data[1].value <<= CORBA::Double(175.00);
ev.filterable_data[2].name    = CORBA::string_dup("CustomerId");
ev.filterable_data[2].value <<= CORBA::Long(1234567);
ev.filterable_data[3].name    = CORBA::string_dup("CustomerName");
ev.filterable_data[3].value <<= "Jane Jones";

// Java
StructuredEvent ev;
...
ev.filterable_data[0].name = "StockName";
ev.filterable_data[0].value.insert_string("BEAS");
ev.filterable_data[1].name = "PricePerShare";
ev.filterable_data[1].value.insert_double(175.00);
ev.filterable_data[2].name = "CustomerId";
ev.filterable_data[2].value.insert_long(1234567);
ev.filterable_data[3].name = "CustomerName";
ev.filterable_data[3].value.insert_string("Jane Jones");
```

For more information about filter grammar, see Creating FML Field Table Files for Events and the section "Boolean Expression of fielded Buffers" in the *BEA TUXEDO FML Programmer's Guide*.

push_consumer

Identifies the callback object that will be used by the Notification Service to deliver a structured event. Subscriber applications must implement the CosNotifyComm::StructuredPushConsumer interface so that the Notification Service can call it to deliver events.

Note: You can use either transient or persistent object references for the callback objects. Both QoS and application run times should be taken into consideration when deciding which type of object reference to use. For information to assist you in deciding which type of object reference to use, refer to Table 2-3.

Table 2-3 When to Use Transient Versus Persistent Object References for Joint Client/Servers

If the subscription ...	Then ...
Will have a transient QoS and will start and shutdown once.	You should use a transient object reference. It this case, we recommend the subscriber application unsubscribe on shutdown so as to release system resources, however, this is not a requirement.
Will have a persistent QoS and will start and shutdown once.	You should use a transient object reference.
Will have a persistent Qos and will start and shutdown multiple times.	You must use a persistent object reference and store the host and port so the same host and port is used each time the subscriber shuts down and restarts. In this case, use of the bidirectional IIOP feature is not recommended. **Note:** If a joint client/server is used, it must be remote (outside the WLE domain) because persistent object references are not supported

	inside the domain.
Will have a transient QoS and will start and shutdown multiple times.	You can use a persistent object reference; however, we do not recommend this configuration unless you can guarantee that no events for this subscriber will be posted while the subscriber is shut down.

qos (quality of service)

Specifies the desired quality of service of the subscription. It can take one of two values: transient or persistent.

For transient subscriptions, the Notification Service makes only one attempt to the deliver the event to a subscriber. If that attempt fails, the event is discarded and, if the Notification Service does not receive the CORBA::TRANSIENT exception, it concludes that the subscriber is shutdown or otherwise not available and cancels the subscription. If the Notification Service receives the CORBA::TRANSIENT exception when an attempt to deliver fails, it assumes that the subscriber is busy and discards the event, but it does not cancel the subscription.

For persistent subscriptions, if the first delivery attempt fails, the Notification Service holds the event in the pending queue and keeps attempting to deliver the subscription until the configurable retry limit is reached. When the retry limit is reached, the Notification Service moves the event on an error queue where it is held for disposition by the system administrator. The administrator either removes the event from the error queue, which in effect discards it, or moves it back to the pending queue so that further attempts to deliver it can be made.

Note: For persistent subscriptions, the Notification Service always does a two-way invoke on callback objects to deliver events. If a joint client/server does not activate a callback object (the event receiver) before it calls orb->run and then the Notification Service invokes on the callback object, as far as the POA is concerned, the callback object does not exist. In this case CORBA::OBJECT_NOT_EXIST exception is returned. If the Notification Service receives a CORBA::OBJECT_NOT_EXIST exception, it drops the subscription and the event; otherwise, the subscription is retained and the event is retried.

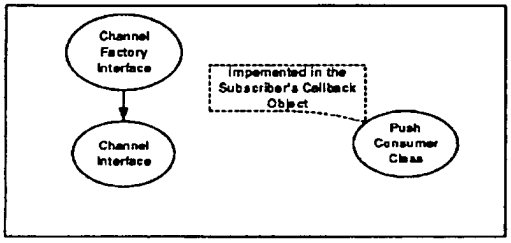
BEA Simple Events API

Simplicity and ease-of-use are the defining characteristics of the BEA Simple Events application programming interface (API). Its capabilities are similar to those of the BEA TUXEDO Event Broker.

The BEA Simple Events API consists the following interfaces (see Figure 2-2):

- Tobj_SimpleEvents::Channel
- Tobj_SimpleEvents::ChannelFactory
- CosNotifyComm::StructuredPushConsumer

Figure 2-2 BEA Simple Events Interfaces



The Tobj_SimpleEvents::Channel and the Tobj_SimpleEvents::ChannelFactory interfaces are implemented by the Notification Service and are described below.

The CosNotifyComm::StructuredPushConsumer interface is implemented by the subscribers. For a description of this interface, see CosNotifyComm::StructuredPushConsumer::push_structured_event.

Note: The CosNotification Service classes referred to in this section are fully described in the CosNotification Service IDL files, which are located in the wledir/include directory.

Note: If you use class operations that are not supported, the CORBA::NO_IMPLEMENT exception is raised.

TOBJ_SimpleEvents::Channel Interface

The Channel interface is used:

- By subscribers to subscribe and unsubscribe to events and to determine if a subscription exists
- By posters to post events to the Notification Service

This interface provides these operations:

- subscribe()
- unsubscribe()
- exists()
- push_structured_event()

The CORBA IDL for this interface:

```
module Tobj_SimpleEvents
 {
 typedef long   SubscriptionID;
 typedef string RegularExpression;
 typedef string FilterExpression;

 const SubscriptionType TRANSIENT_SUBSCRIPTION  = 0;
 const SubscriptionType PERSISTENT_SUBSCRIPTION = 1;

interface Channel
  {
   void push_structured_event(
      in  CosNotification::StructuredEvent   event);

   SubscriptionID subscribe (
     in   string                    subscription_name,
     in   RegularExpression             domain,
     in   RegularExpression             type,
     in   FilterExpression            data_filter,
     in   CosNotification::QoSProperties   qos,
     in   CosNotifyComm::StructuredPushConsumer push_consumer);

   boolean exists( in SubscriptionID id );

   void unsubscribe( in SubscriptionID  id );
  };
 };
```

These operations are described in the following section.

channel::subscribe

CORBA IDL

```
SubscriptionID subscribe (
    in   string                    subscription_name,
    in   RegularExpression             domain,
    in   RegularExpression             type,
    in   FilterExpression            data_filter,
    // The filter expression must length 1 and the name must
    // be TRANSIENT_SUBSCRIPTION or PERSISTENT_SUBSCRIPTION.
    in   CosNotification::QoSProperties   qos,
    in   CosNotifyComm::StructuredPushConsumer push_consumer
);
```

Exceptions

CORBA::BAD_PARAM

Indicates one of the following problems:
Tobj_Events::SUB_INVALID_FILTER_EXPRESSION
Tobj_Events::SUB_UNSUPPORTED_QOS_VALUE

CORBA::IMP_LIMIT

Indicates one of the following problems:
Tobj_Events::SUB_DOMAIN_BEGINS_WITH_SYSEV

Tobj_Events::SUB_EMPTY_DOMAIN

Tobj_Events::SUB_EMPTY_TYPE

Tobj_Events::SUB_DOMAIN_AND_TYPE_TOO_LONG

Tobj_Events::SUB_FILTER_TOO_LONG

Tobj_Events::SUB_NAME_TO_LONG

Tobj_Events::TRANSIENT_ONLY_CONFIGURATION

CORBA::INV_OBJREF

Indicates the following problem:

Tobj_Events::SUB_NIL_CALLBACK_REF

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Description:

Use this operation to subscribe to events. This operation is called by a subscriber application on the Notification Service to create a subscription to a particular event. The subscription name, domain name, type name, data filter, quality of service, and the object reference of the subscriber's callback object are passed in. The callback object implements the CosNotifyComm::StructuredPushConsumer IDL interface.

Note: For subscribers that shut down and restart, you must write the subscription_id to persistent storage.

To use data filtering or subscribe to BEA TUXEDO system events or events posted by a BEA TUXEDO application, see the sections Creating FML Field Table Files for Events and Interoperability with BEA TUXEDO Applications

Parameters

For a description of the parameters supported by this operation, see Parameters Used When Creating Subscriptions.

Return Value

Returns a unique subscription identifier. The effect of this operation is not instantaneous. There can be a delay between returning from this operation and the actual start of event delivery. The length of the delay period may be significant depending on your configuration. For more information on factors impacting this delay period, see Synchronizing Databases.

Note: Notification Service applications that start and shut down only once can use the subscription_id to determine if their subscription has been cancelled automatically or by the system administrator.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating a Subscription.

C++ code example:

```
subscription_id = channel->subscribe(
        subscription_name,
        "News", // domain
        "Sports",   // type
        "",     // No data filter.
        qos,
        news_consumer.in()
      );
```

Java code example:

```
int subscription_id = channel.subscribe(
        subscription_name,
        "News", // domain
        "Sports",   // type
        "",     // no data filter
        qos,
        news_consumer_impl
      );
```

Channel::unsubscribe

CORBA IDL

```
void unsubscribe( in SubscriptionID id );
```

Parameter

subscription_id

The subscription identifier.

Exceptions

CORBA::BAD_PARAM

Indicates the following problem:

Tobj_Events::INVALID_SUBSCRIPTION_ID

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Description:

Used to unsubscribe. Subscriber applications use this operation to terminate subscriptions. On return from this operation, no further events can be delivered. There is one input parameter: SubscriptionID , which you got when you subscribed.

Note: This operation is not instantaneous. After returning from this operation, a subscriber may continue to receive events for a period of time. The period of time may be significant depending on your configuration. For more information on factors impacting this period of time, see Synchronizing Databases.

Examples

C++ code example:

```
channel->unsubscribe(subscription_id);
```

Java code example:

```
channel.unsubscribe(subscription_id);
```

Channel::push_structured_event

CORBA IDL

```
void push_structured_event(
    in     CosNotification::StructuredEvent   notification
);
```

Exceptions

CORBA_IMP_LIMIT

Indicates one of the following problems with the subscription:

Tobj_Events::POST_UNSUPPORTED_VALUE_IN_ANY

Tobj_Events::POST_UNSUPPORTED_PRIORITY_VALUE

Tobj_Events::POST_DOMAIN_CONTAINS_SEPARATOR

Tobj_Events::POST_TYPE_CONTAINS_SEPARATOR

Tobj_Events::POST_SYSTEM_EVENTS_UNSUPPORTED

Tobj_Events::POST_EMPTY_DOMAIN

Tobj_Events::POST_EMPTY_TYPE

Tobj_Events::POST_DOMAIN_AND_TYPE_TOO_LONG

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Parameter

notification

This parameter contains the structured event as defined by the CosNotification Service specification.

Description:

Used by the poster application to post an event to the Notification Service.

Note: This operation has transactional behavior when used in the context of a transaction. For more information, see the section Using Transactions.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating and Posting Events.

C++ code example:

```
channel->push_structured_event(notification);
```

Java Code example:

```
channel.push_structured_event(notification);
```

Channel::exists

CORBA IDL

```
boolean exists(in SubscriptionID subscription_id);
```

Parameter

subscription_id

The subscription identifier.

Exceptions

CORBA::BAD_PARAM

Indicates the following problem: Tobj_Events::INVALID_SUBSCRIPTION_ID

If the subscription_id is for a subscription created using the CosNotification Service API, this exception is always returned.

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Description:

Used by subscriber applications to determine if a subscription exists. Since the system administrator can delete subscriptions manually and the Notification Service can delete transient subscriptions automatically, a subscriber application might want to use this operation so that it can re-create the subscription, if necessary. The subscription_id used in this operation is the same one that you got when you subscribed.

Return Value

Returns Boolean True of the subscription exists and False if it does not.

Examples

C++ code example:

```
if channel->exists (subscription_id) {
   // The subscription is still valid.
} esle {
  // The subscription no longer exists.

}
```

Java code example:

```
if channel.exists (subscription_id) {
   // The subscription is still valid.
} esle {
  // The subscription no longer exists.

}
```

TOBJ_SimpleEvents::ChannelFactory Interface

The ChannelFactory interface is used to find event channels. This interface provides a single operation: find_channel .

The CORBA IDL for this interface:

```
module Tobj_SimpleEvents
{
  typedef long   ChannelID;

interface ChannelFactory
{
 Channel find_channel(
   in ChannelID channel_id // Must be DEFAULT_CHANNEL
  );
 };
```

};

Channel_Factory::find_channel

CORBA IDL

```
Channel find_channel(
    in ChannelID channel_id );
```

Parameter

In this release of WLE, there can only be one event channel; therefore, the ChannelID that is passed in must be set to Tobj_SimpleEvents::DEFAULT_CHANNEL (for C++) or Tobj_SimpleEvents.DEFAULT_CHANNEL.Value (for Java).

Exceptions

CORBA::BAD_PARAM

Indicates the following problem:

Tobj_Events::INVALID_CHANNEL_ID

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Description:

Used by poster applications and subscriber applications. This operation is used to find the event channel so that it can be used by the poster to post events and by the subscriber to subscribe and unsubscribe to events.

Return Value

Returns the default event channel's object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Getting the Event Channel.

C++ code example:

```
channel_factory->find_channel(
        Tobj_SimpleEvents::DEFAULT_CHANNEL);
```

Java code example:

```
channel_factory.find_channel(DEFAULT_CHANNEL.value);
```

CosNotification Service API

This section contains a discussion of the operations defined by the CosNotification Service that are implemented by WLE Notification Services. These operations are only a subset of the complete set of operations. This subset is a functionally complete API that can be used as an alternative to the BEA Simple Events API.

This API is necessarily more complex then the BEA Simple Events API. There are two reasons for this. First, the CosNotification Service API is more complex. Second, the WLE implementation of the CosNotification Service API places additional restrictions on the operations that are supported. Because this complexity offers no advantages in terms of performance or flexibility, we recommend that you use the BEA Simple Events API whenever possible.

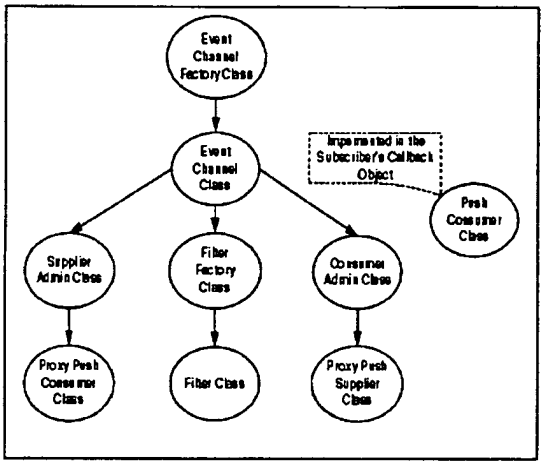
The CosNotification API is provided for those who require that a standard API be used

whenever possible for purposes of portability. As regards functionality, this API provides no benefits beyond those offered by the Simple Events API. Applications that are developed using this API will be mostly, but not completely, portable. The reason for this is that not enough of the CosNotification Service API is supported to facilitate portability. For example, the filtering grammar required by CORBA-based Notification Service is based on the COS Trader grammar. Since WLE does not support this grammar, but supports an alternative grammar based on the BEA TUXEDO Event Broker grammar, any application that requires filtering will not be portable. The same is true for QoS, that is, the CosNotification Service API does not support the CORBA-based Notification Service standard qualities of service, but it does support alternative qualities of service.

Overview of Supported CosNotification Service Classes

Figure 2-3 shows the CosNotification Service classes implemented in full or in part in this release of WLE and their relationships.

Figure 2-3 Implemented CosNotification Service Classes



The operations supported by each class are summarized below. For more detailed descriptions, see Detailed Descriptions of CosNotification Service Classes

- CosNotifyChannelAdmin::EventChannelFactory Class

  This class is used by the event poster and subscriber applications. It supports the get_channel_factory operation which is used to get the channel factory when posting, subscribing, and unsubscribing to events.

- CosNotifyChannelAdmin::EventChannel Class

  This class is used by event poster and subscriber applications. It supports three operations:

  - default_consumer_admin -used by event subscriber applications to get the consumer admin object.
  - default_supplier_admin -used by event poster applications to get the supplier admin object.
  - default_filter_factory -used by event subscriber applications to get the filter factory object.

- CosNotifyChannelAdmin::SupplierAdmin Class

  This class is used by event poster applications. It supports the obtain_notification_push_consumer operation. Poster applications use this operation to create proxy push consumer objects which in turn are used to post events to the Notification Service.

- CosNotifyChannelAdmin::StructuredProxyPushConsumer Class

  This class is used by event poster applications. It supports the following operations:

- connect_structured_push_supplier -used by event poster applications to connect the proxy push supplier to the Notification Service event channel.
- push_structured_event -used by event poster applications to post the event to the Notification Service event channel.
- disconnect_structured_push_consumer -used by event poster applications to disconnect the proxy push supplier from the Notification Service event channel.

- CosNotifyFilter::FilterFactory Class

  This class is used by event subscriber applications to create a filter object. It supports the create_filter operation. The filter object provides all data filtering including domain, type, and filterable data.

- CosNotifyFilter::Filter Class

  This class is used by event subscriber applications. It supports the following operations:

  - add_contraints operation-used to set the filter's domain, type, and data filter.
  - destroy operation-used to destroy the filter object.

- CosNotifyChannelAdmin::ConsumerAdmin Class

  This class is used by event subscriber applications. It supports the following operations:

  - obtain_notification_push_supplier -used by event subscriber applications to create proxy push supplier objects which in turn are used to deliver events to the subscriber's callback object.
  - get_proxy_supplier -used by event subscriber applications to retrieve the object reference for the proxy push supplier object. This operation is only used when the subscriber application shuts down then restarts and cancels the subscription. This is because subscribers need to discard the object reference from the first run and get it back again for the next run. Subscribers cannot reuse object references from one run to the next.

- CosNotifyChannelAdmin::StructuredProxyPushSupplier Class

  This class is used by event subscriber applications. It supports the following operations:

  - connect_structured_push_consumer -used by event subscriber applications to connect the subscriber to the proxy push supplier.
  - set_qos -used by event subscriber applications to set the quality of service for subscriptions.
  - add_filter -used by event subscriber applications to add the filter object to the subscription.
  - get_filter -used by event subscriber applications when performing unsubscribe operations to get the filter associated with the subscription. This operation is only used when the subscriber application shuts down then restarts.
  - disconnect_structured_push_supplier -used by event subscriber applications to unsubscribe.

- CosNotifyComm::StructuredPushConsumer

  This interface is implemented by event subscriber applications. It supports the push_structured_event operation. The Notification Service invokes this operation to deliver events to the subscriber.

Detailed Descriptions of CosNotification Service Classes

This section describes the CosNotification Service classes that this release of WLE implements. These classes are fully described in the CosNotification Service IDL files, which are located in the wledir/include directory.

Note: If you use class operations that are not supported, the CORBA::NO_IMPLEMENT exception is raised.

CosNotifyFilter::Filter Class

This class is used by event subscriber applications. The OMG IDL for this class is as

follows:

```
Module CosNotifyFilter
{
interface Filter {
    ConstraintInfoSeq add_constraints (
          in ConstraintExpSeq constraint)
        raises (InvalidConstraint);

     void destroy();
   };
}; //CosNotifyFilter
```

CosNotifyFilter::Filter::add_constraints

Synopsis

Sets the domain, type, and data filter parameters on the filter object.

OMG IDL

```
ConstraintInfoSeq add_constraints (
        in ConstraintExpSeq constraint)
      raises (InvalidConstraint);
```

Exceptions

CosNotifyFilter::InvalidConstraint

Never raised.

CORBA::BAD_PARAM

Indicates the following problem: Tobj_Events::SUB_INVALID_FILTER_EXPRESSION.

CORBA_IMP_LIMIT

Indicates one of the following problems:

Tobj_Notification::SUB_ADD_CONS_ON_TIMED_OUT_FILTER

Tobj_Notification::SUB_MULTIPLE_CALLS_TO_ADD_CONS

Tobj_Notification::SUB_MULTIPLE_CONSTRAINTS_IN_LIST

Tobj_Notification::SUB_MULTIPLE_TYPES_IN_CONSTRAINT

Tobj_Notification::SUB_SYSTEM_EVENTS_UNSUPPORTED

Tobj_Events::SUB_DOMAIN_BEGINS_WITH_SYSEV

Tobj_Events::SUB_EMPTY_DOMAIN

Tobj_Events::SUB_EMPTY_TYPE

Tobj_Events::SUB_FILTER_TOO_LONG

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Description

Used when subscribing. This operation is used in subscriber applications to define the kind of event to which you want to subscribe. You set the domain, type, and data filter parameters on the filter object. For a description of these parameters, see Parameters Used When Creating Subscriptions.

Note: The WLE implementation of the add_constraints operation 1) can only be called once, 2) must be called before the filter is added to the proxy object, and 3) must consist of

only a single constraint which has a single event type.

Return Value

Returns an empty list , which we recommend that the caller ignores.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating a Subscription.

C++ code example:

```
// set the filtering parameters
// (domain = "News", type, and no data filter)
CosNotifyFilter::ConstraintExpSeq constraints;
constraints.length(1);
constraints[0].event_types.length(1);
constraints[0].event_types[0].domain_name =
   CORBA::string_dup("News");
constraints[0].event_types[0].type_name =
          CORBA::string_dup ("Sports");
// no data filter
constraints[0].constraint_expr = CORBA::string_dup("");          CosNotifyFilter::ConstraintInfoSeq_var
add_constraints_results = // ignore this returned value
   filter->add_constraints(constraints);
```

Java code example:

```
// set the filtering parameters
// (domain = "News", type, and no data filter).
ConstraintExp constraints[] = new ConstraintExp[1];
constraints[0] = new ConstraintExp();
constraints[0].event_types = new EventType[1];
constraints[0].event_types[0] = new EventType();
constraints[0].event_types[0].domain_name = "News";
constraints[0].event_types[0].type_name = "Sports";
constraints[0].constraint_expr = ""; // No data filter.
ConstraintInfo add_constraints_results[] =
  filter.add_constraints(constraints); //Ignore this return value.
```

CosNotifyFilter::Filter::destroy

Synopsis

Destroys the filter object.

OMG IDL

void **destroy** ();

Exceptions

CORBA::BAD_PARAM

Indicates the following problem: Tobj_Events::SUB_INVALID_FILTER_EXPRESSION.

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Description

Used when unsubscribing. This operation is used in subscriber applications to destroy the target filter object.

Note: Do not destroy the filter object until you are ready to cancel the corresponding subscription.

CosNotifyFilter::FilterFactory Class

This class is used by event subscriber applications. The OMG IDL for this class is as follows:

```
Module CosNotifyFilter
{
interface FilterFactory {
  Filter create_filter (
       in string constraint_grammar)
    raises (InvalidGrammar);
  destroy();
 };
}; //CosNotifyFilter
```

CosNotifyFilter::FilterFactory::create_filter

Synopsis

Determines which events are delivered to a subscription.

OMG IDL

```
Filter create_filter (
in string constraint_grammar)
raises (InvalidGrammar);
```

Exceptions

CosNotifyFilter::InvalidGrammar

Indicates the constraint_grammar is not supported.

Description

Used in the subscriber application to create a new filter object. This filter is used to determine which events are delivered to a subscription. The subscriber must set up the filter and add it to the proxy within five minutes; otherwise, the filter will be destroyed. The filter grammar must be set to Tobj_Notification::Constraint_grammar ; otherwise, the InvalidGrammar exception is raised.

Return Value

Returns the new filter's object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating a Subscription.

C++ code example:

```
filter_factory->create_filter(
    Tobj_Notification::CONSTRAINT_GRAMMAR
);
```

Java code example

```
filter_factory.create_filter(CONSTRAINT_GRAMMAR.value);
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier Class

This class is used by event subscriber applications. The OMG IDL for this class is as

follows:

```
Module CosNotifyChannelAdmin
{
 interface StructuredProxyPushSupplier :
     ProxySupplier,
     CosNotifyComm::StructuredPushSupplier {

   void connect_structured_push_consumer (
        in CosNotifyComm::StructuredPushConsumer push_consumer)
      raises(CosEventChannelAdmin::AlreadyConnected,
           CosEventChannelAdmin::TypeError );
   };
   // The following operations are inherited.
   void set_qos(in QoSProperties qos)
      raises (UnsupportedQoS);
   FilterID add_filter (in Filter new_filter );
   Filter get_filter( in FilterID filter )
      raises ( FilterNotFound);
   void disconnect_structured_push_supplier();
   readonly attribute ProxyType     MyType;
  };
}; //CosNotifyChannelAdmin
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier:: connect_structured_push_consumer

Synopsis

Completes a subscription.

OMG IDL

void **connect_structured_push_consumer** (
in CosNotifyComm::StructuredPushConsumer push_consumer)
raises(CosEventChannelAdmin::AlreadyConnected,
CosEventChannelAdmin::TypeError);

Exceptions

CosEventChannelAdmin::TypeError

Never raised.

CORBA::INV_OREF

Tobj_Events::SUB_NIL_CALLBACK_REF

CORBA::IMP_LIMIT

Indicates one of the following problems:

Tobj_Events::SUB_DOMAIN_AND_TYPE_TOO_LONG

Tobj_Events::SUB_NAME_TO_LONG

Tobj_Events::TRANSIENT_ONLY_CONFIGURATION

Tobj_Notification::SUBSCRIPTION_DOESNT_EXIST.

CORBA::OBJECT_NOT_EXIST

The proxy does not exist.

CosEventChannelAdmin::AlreadyConnected

Indicates that the **connect_structured_push_consumer** operation has already been invoked.

Note: For exception definitions and corresponding minor codes, see Exception Minor Codes.

Description

Use this operation when subscribing. This operation is used in subscriber applications to subscribe to events. The push_consumer parameter identifies the subscriber's callback object.

Once the **connect_structured_push_consumer** has been called, the Notification Service will proceed to send events to the subscriber by invoking the callback object's push_structured_event operation. If the **connect_structured_push_consumer** has already been called, the AlreadyConnected exception is raised.

Note: You must call set_qos and add_filter before calling **connect_structured_push_consumer** .

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating a Subscription.

C++ code example:

```
subscription->connect_structured_push_consumer(
   news_consumer.in()
);
```

Java code example:

```
subscription.connect_structured_push_consumer(
     news_consumer_impl
);
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier::s

Synopsis

Sets the QoS for the subscription.

OMG IDL

```
void set_qos(in QoSProperties qos)
raises (UnsupportedQoS);
```

Exceptions

UnsupportedQoS

Never raised.

ORBA::IMP_LIMIT

Indicates one of the following problems:

Tobj_Notification::SUB_MULTIPLE_CALLS_TO_SET_QOS

Tobj_Notification::SUB_CANT_SET_QOS_AFTER_CONNECT

Tobj_Notification::SUBSCRIPTION_DOESNT_EXIST

Tobj_Notification::SUB_UNSUPPORTED_QOS_VALUE

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Description

Used when subscribing. This operation is used in subscriber applications to set the QoS for the subscription. It takes as an input parameter a sequence of name-value pairs which encapsulates quality of service property settings that the subscriber is requesting.

There are two components of the QoS, the subscription type and the subscription name. The subscription type is set by constructing a name-value pair where the name is Tobj_Notification::SUBSCRIPTION_TYPE and the value is either Tobj_Notification::PERSISTENT_SUBSCRIPTION , or Tobj_Notification::TRANSIENT_SUBSCRIPTION . Further explanation, and additional usage details, see Quality of Service.

The subscription name is set by constructing a name-value pair, where the name is Tobj_Notification::SUBSCRIPTION_NAME , and the value is a user defined string.

For more information on this parameter, see Parameters Used When Creating Subscriptions.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating a Subscription.

C++ code example:

```
CosNotification::QoSProperties qos;
qos.length(2);
qos[0].name =
   CORBA::string_dup(Tobj_Notification::SUBSCRIPTION_NAME);
qos[0].value <<= "MySubsription";
qos[1].name =
   CORBA::string_dup(Tobj_Notification::SUBSCRIPTION_TYPE);
qos[1].value <<=
   Tobj_Notification::TRANSIENT_SUBSCRIPTION;

subscription->set_qos(qos);
```

Java code example:

```
Property qos[] = new Property[2];
qos[0] = new Property();
qos[0].name = SUBSCRIPTION_NAME.value;
qos[0].value = orb.create_any();
qos[0].value.insert_string("MySubsription");
qos[1] = new Property();
qos[1].name = SUBSCRIPTION_TYPE.value;
qos[1].value = orb.create_any();
qos[1].value.insert_short(TRANSIENT_SUBSCRIPTION.value);

subscription.set_qos(qos);
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier::a

Synopsis

Sets the filter object on the subscriber's callback object.

OMG IDL

```
add_filter(
    in Filter new_filter
);
```

Exceptions

CORBA::IMP_LIMIT

Indicates one of the following problems:

Tobj_Notification::SUB_MULTIPLE_CALLS_TO_SET_FILTER

Tobj_Notification::SUB_ADD_FILTER_AFTER_CONNECT

Tobj_Notification::SUB_NIL_FILTER_REF

Tobj_Notification::SUB_NO_CUSTOM_FILTERS

CORBA::OBJECT_NOT_EXIST

Indicates that the subscription does not exist.

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Description

Used when subscribing. This operation is used in subscriber applications to set the filter object to the subscriber's callback object. If the application using this operation will be shut down and restarted, the filter_id should be written to persistent storage.

Note: This operation 1) cannot be called after the subscriber callback object is connected (see connect_structured_push_consumer above), 2) cannot be called more than once, and 3) when it is called, the filter constraint expression must already be present in the filter (see CosNotifyFilter::Filter add_constraints).

Note: Only filters created by the event channel's default filter factory can be added.

Return Value

Returns a filter_id .

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating a Subscription.

C++ code example:

```
CosNotifyFilter::FilterID filter_id =
    subscription->add_filter(filter.in());
```

Java code example:

```
int filter_id = subscription.add_filter(filter);
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier::g

Synopsis

Gets an object reference to the filter currently associated with the subscriber's callback object.

OMG IDL

```
Filter get_filter( in FilterID filter )
    raises ( FilterNotFound);
```

Exceptions

CosNotifyChannelAdmin::FilterNotFound

The filter could not be found.

Description

Used when a restartable subscriber wants to unsubscribe. This operation is used in subscriber applications to get an object reference to the filter currently associated with the subscriber's callback object. The FilterID that is passed in must be valid for the subscriber's StructuredProxyPushSupplier object. If the FilterID is not valid for any proxy object associated with the event channel, then a FilterNotFound exception is thrown. The operation is only used by subscribers that shut down and restart.

Restrictions

The following usage restrictions and guidelines apply to this operation:

a. Filter object references that are returned from this operation cannot be used in comparison operations.

b. Filter object references returned by this operation can be used by the CosNotifyFilter::Filter::destroy operations but are of little use otherwise since they can not be modified or added to proxy objects.

Return Value

Returns a filter object reference to the filter currently associated with the subscriber's callback object.

Examples

C++ code example:

```
CosNotify::Fitler_var filter =
   subscription->get_filter( filter_id() );
```

Java code example:

```
Filter filter = subscription.get_filter( filter_id() );
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier:: disconnect_structured_push_supplier

Synopsis

Used to unsubscribe.

OMG IDL

```
void disconnect_structured_push_supplier();
```

Exceptions

CORBA::OBJECT_NOT_EXIST

Indicates that the subscription to be disconnected does not exist.

Note: For more information on exceptions and corresponding minor codes, see Exception Minor Codes.

Description

Used by subscriber applications when unsubscribing. This operation is used in subscriber applications to terminate a connection between the Notification Service and the subscriber's callback object.

Note: This operation does not stop event delivery instantaneously. After returning from this operation, a subscriber may continue to receive events for a period of time.

Examples

C++ code example:

```
subscription->disconnect_structured_push_supplier();
```

Java code example:

```
subscription.disconnect_structured_push_supplier();
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier::M

Synopsis

Always returns CosNotifyChannelAdmin::PUSH_STRUCTURED proxy.

OMG IDL

```
readonly attribute ProxyType MyType
```

Description

Always returns CosNotifyChannelAdmin::PUSH_STRUCTURED proxy.

CosNotifyChannelAdmin::StructuredProxyPushConsumer Class

This class is used by event posting applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface StructuredProxyPushConsumer :
      ProxyConsumer,
      CosNotifyComm::StructuredPushConsumer {

   void connect_structured_push_supplier (
        in CosNotifyComm::StructuredPushSupplier push_supplier)
     raises(CosEventChannelAdmin::AlreadyConnected);
   // The following operations are inherited.
   readonly attribute MyType;
   void push_structured_event(
        in CosNotification::StructuredEvent notification )
     raises( CosEventComm::Disconnected );
   void disconnect_structured_push_consumer();
  };
}; \\StructuredProxyPushConsumer
```

CosNotifyChannelAdmin::StructuredProxyPushConsumer::connect_structured_push_supplier

Synopsis

Prepares the Notification Service to receive an event.

OMG IDL

```
void connect_structured_push_supplier (
       in CosNotifyComm::StructuredPushSupplier push_supplier)
     raises(CosEventChannelAdmin::AlreadyConnected);
```

Exception

CosEventChannelAdmin::AlreadyConnected

Never raised.

Description

Used by poster applications when posting events. You must call this operation to prepare the Notification Service to receive an event and you must pass in a NIL when you use this operation. The sequence of usage is as follows:

1. Make a proxy.
2. Use this operation to connect to the Notification Service and pass in a NIL.
3. Post events.
4. Before exiting the poster program, disconnect.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating and Posting Events.

C++ code example:

```
proxy_push_consumer->connect_structured_push_supplier(
    CosNotifyComm::StructuredPushSupplier::_nil()
  );
```

Java code example:

```
proxy_push_consumer.connect_structured_push_supplier(null);
```

CosNotifyChannelAdmin::StructuredProxyPushConsumer: push_structured_event

Synopsis

Posts events to the event channel.

OMG IDL

```
void push_structured_event(
      in CosNotification::StructuredEvent notification )
    raises( CosEventComm::Disconnected );
```

Exceptions

CosEventComm::Disconnected

Never raised.

CORBA::IMP_LIMIT

Indicates one of the following problems:

Tobj_Events::POST_UNSUPPORTED_VALUE_IN_ANY

Tobj_Events::POST_UNSUPPORTED_PRIORITY_VALUE

Tobj_Events::POST_DOMAIN_CONTAINS_SEPARATOR

Tobj_Events::POST_TYPE_CONTAINS_SEPARATOR

Tobj_Events::POST_SYSTEM_EVENTS_UNSUPPORTED

Tobj_Events::POST_EMPTY_DOMAIN

Tobj_Events::POST_EMPTY_TYPE

Tobj_Events::POST_DOMAIN_AND_TYPE_TOO_LONG

Note: For more information on exceptions and corresponding minor codes, see Exception

Minor Codes.

Descriptions

Used when posting events. This operation is used in poster applications to post events to the event channel.

Note: This operation differs from the standard CORBA definition in the following ways:
1) The Priority in the variable header section of the event, if specified, must be short value in the range of 1 to 100.
2) If event filterable data filtering (versus filtering on domain and type only) is required, or if events are to be received by a BEA TUXEDO subscriber, then additional restrictions apply. See Structured Event Fields, Types and Filters and Interoperability with BEA TUXEDO Applications.

Note: This operation has transactional behavior when used in the context of a transaction. For more information, see Using Transactions.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating and Posting Events.

C++ code example:

```
proxy_push_consumer->push_structured_event(notification);
```

Java code example:

```
proxy_push_consumer.push_structured_event(notification);
```

CosNotifyChannelAdmin::StructuredProxyPushConsumer: disconnect_structured_push_consumber

Synopsis

Stops posting events.

OMG IDL

```
void disconnect_structured_push_consumer();
```

Descriptions

Used when posting events. This operation is used by poster applications to stop posting events. It takes no input parameters and returns no values. The recommended usage sequence is as follows:

1. Make a proxy.
2. Connect and disconnect on every run of the poster application.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating and Posting Events.

C++ code example:

```
proxy_push_consumer->disconnect_structured_push_consumer();
```

Java code example:

```
proxy_push_consumer.disconnect_structured_push_consumer();
```

CosNotifyChannelAdmin::StructuredProxyPushconsumer::

Synopsis

Always returns CosNotifyChannelAmdmin::PUSH_STRUCTURED proxy.

Synopsis

readonly attribute ProxyType MyType

Description

Always returns CosNotifyChannelAmdmin::PUSH_STRUCTURED proxy.

CosNotifyChannelAdmin::ConsumerAdmin Class

This class is used by event subscriber applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface ConsumerAdmin :

      CosNotification::QoSAdmin,
      CosNotifyComm::NotifySubscribe,
      CosNotifyFilter::FilterAdmin,
      CosEventChannelAdmin::ConsumerAdmin {

    ProxySupplier obtain_notification_push_supplier (
        in ClientType ctype,
        out ProxyID proxy_id)
      raises ( AdminLimitExceeded )

    ProxySupplier get_proxy_supplier (
        in ProxyID proxy_id )
      raises ( ProxyNotFound );

 };
}; //CosNotifyChannelAdmin
```

CosNotifyChannelAdmin::ConsumerAdmin:: obtain_notification_push_supplier

Synopsis

Creates proxy push supplier objects.

OMG IDL

```
ProxySupplier obtain_notification_push_supplier (
      in ClientType ctype,
      out ProxyID proxy_id)
    raises ( AdminLimitExceeded )
```

Exceptions

CosNotifyChannelAdmin::AdminLimitExceeded

Never raised.

CORBA::IMP_LIMIT

Indicates the following problem:

Tobj_Notification::SUB_UNSUPPORTED_CLIENT_TYPE

Description

Used when subscribing. This operation is used in subscriber applications to create proxy push supplier objects. Only structured events are supported (that is, ANY_EVENT and SEQUENCE_EVENT ClientTypes are not supported). Therefore, the ClientType input parameter must be set to CosNotifyComm::STRUCTURED_EVENT . If you shut down and restart the subscriber and subscription survives more than one run of you program, the ProxyID returned by this operation should be durably stored. The subscriber must narrow the proxy supplier to CosNotifyChannelAdmin::StructuredProxyPushSupplier . All required operations must be completed in five minutes.

Note: Notification Service applications that start and shut down only once can use the proxy_id to determine if their subscription has been cancelled automatically or by the system administrator.

Return Value

This operation returns the new proxy's object reference. The new proxy_id is also returned through the proxy_id out parameter.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating a Subscription.

C++ code example:

```
CosNotifyChannelAdmin::ProxySupplier_var generic_proxy =
    consumer_admin->obtain_notification_push_supplier(
      CosNotifyChannelAdmin::STRUCTURED_EVENT,
      proxy_id
    );

CosNotifyChannelAdmin::StructuredProxyPushSupplier_var proxy =
CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(
generic_proxy.in ()
);
```

Java code example:

```
ProxySupplier generic_proxy =
    consumer_admin.obtain_notification_push_supplier(
      ClientType.STRUCTURED_EVENT,
      proxy_id
    );

   StructuredProxyPushSupplier proxy =
    StructuredProxyPushSupplierHelper.narrow(
     generic_proxy
    );
```

CosNotifyChannelAdmin::ConsumerAdmin::get_proxy_sup

Synopsis

Returns the proxy push supplier object created using the consumer admin object **obtain_notification_push_supplier** operation.

OMG IDL

```
ProxySupplier get_proxy_supplier (
    in ProxyID proxy_id )
  raises ( ProxyNotFound );
```

Exceptions

CosNotifyChannelAdmin::ProxyNotFound

Indicates that the ProxyID could not be found.

Descriptions

Used when unsubscribing. This operation is used in subscriber applications to return the proxy push supplier object created using the consumer admin object **obtain_notification_push_supplier** operation. The ProxyID input parameter uniquely identifies the proxy object. Callers should be aware that the proxy object can be destroyed either due to an error in delivering a transient subscription or through of an ntsadmin administrative command. When a proxy object is destroyed, the ProxyID associated with it is invalidated. If the ProxyID is invalid a ProxyNotFound exception is raised. The subscriber must narrow the proxy supplier to CosNotifyChannelAdmin::StructuredProxyPushSupplier .

Return Value

Returns the object reference for the existing proxy.

Examples

C++ code example:

```
CosNotifyChannelAdmin::ProxySupplier_var generic_proxy =
      m_consumer_admin->get_proxy_supplier(
        m_subscription_info.news_proxy_id()
      );

CosNotifyChannelAdmin::StructuredProxyPushSupplier_var proxy =
    CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(
      generic_proxy.in()
    );
```

Java code example:

```
ProxySupplier generic_subscription =
      m_consumer_admin.get_proxy_supplier(
        m_subscription_info.news_proxy_id()
      );

StructuredProxyPushSupplier subscription =
     StructuredProxyPushSupplierHelper.narrow(
       generic_proxy);
```

CosNotifyChannelAdmin::SupplierAdmin Class

This class is used by event poster applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface SupplierAdmin :
      CosNotification::QoSAdmin,
      CosNotifyComm::NotifyPublish,
      CosNotifyFilter::FilterAdmin,
```

```
    CosEventChannelAdmin::SupplierAdmin {

  ProxyConsumer obtain_notification_push_consumer (
        in ClientType ctype,
        out ProxyID proxy_id)
    raises ( AdminLimitExceeded );
 };
}; //SupplierAdmin
```

CosNotifyChannelAdmin::SupplierAdmin:: obtain_notification_push_consumer

Synopsis

Creates proxy push consumer objects.

C++ Mapping

```
ProxyConsumer obtain_notification_push_consumer (
      in ClientType ctype,
      out ProxyID proxy_id)
    raises ( AdminLimitExceeded );
```

Exceptions

CosNotifyChannelAdmin::AdminLimitExceeded

Never raised.

CORBA::IMP_LIMIT

Indicates the following problem:

Tobj_Notification::SUB_UNSUPPORTED_CLIENT_TYPE

Description

Used when posting events. This operation is used in poster applications to create proxy push consumer objects. ClientType must be set to "CosNotifyChannelAdmin::STRUCTURED_EVENT" . The ProxyID returned should be ignored. The Proxy Consumer must be narrowed the proxy supplier to CosNotifyChannelAdmin::StructuredProxyPushConsumer .

Note: Notification Service applications that start and shut down only once can use the proxy_id to determine if their subscription has been cancelled automatically or by the system administrator.

Return Value

This operation returns the new proxy's object reference. The new proxy_id is also returned through the proxy_id out parameter.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating and Posting Events.

C++ code example:

```
CosNotifyChannelAdmin::ProxyConsumer_var generic_proxy_consumer =
    supplier_admin->obtain_notification_push_consumer(
      CosNotifyChannelAdmin::STRUCTURED_EVENT,
      proxy_id
```

```
    );

  CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
    proxy_push_consumer =
      CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(
       generic_proxy_consumer
      );
```

Java code example:

```
  supplier_admin.obtain_notification_push_consumer(
     ClientType.STRUCTURED_EVENT, proxy_id );
```

CosNotifyChannelAdmin::EventChannel Class

This class is used by event poster applications. The OMG IDL for this class is as follows:

```
  Module CosNotifyChannelAdmin
  {
   interface EventChannel :
       CosNotification::QoSAdmin,
       CosNotification::AdminPropertiesAdmin,
       CosEventChannelAdmin::EventChannel {

      readonly attribute ConsumerAdmin default_consumer_admin;
      readonly attribute SupplierAdmin default_supplier_admin;
      readonly attribute CosNotifyFilter::FilterFactory
                          default_filter_factory;
   };
  }; //CosNotifyChannelAdmin
```

CosNotifyChannelAdmin::EventChannel:: ConsumerAdmin default_consumer_admin

Synopsis

Gets the ConsumerAdmin object.

OMG IDL

```
readonly attribute ConsumerAdmin default_consumer_admin;
```

Description

Used when subscribing and unsubscribing. This operation is used in subscriber applications to get the ConsumerAdmin object.

Return Value

Returns the object reference to the ConsumerAdmin object.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Getting the Event Channel, ConsumerAdmin Object, and Filter Factory Object.

C++ code example:

```
channel->default_consumer_admin();
```

Java code example:

```
channel.default_consumer_admin();
```

CosNotifyChannelAdmin::EventChannel:: ConsumerAdmin default_supplier_admin

Synopsis

Gets the Supplier Admin object.

OMG IDL

```
readonly attribute SupplierAdmin default_supplier_admin;
```

Description

Used when posting events. This operation is used in event poster applications to get the SupplierAdmin object.

Return Value

SupplierAdmin object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Creating and Posting Events.

C++ code example:

```
channel->default_supplier_admin();
```

Java code example:

```
channel.default_supplier_admin();
```

CosNotifyChannelAdmin::EventChannel::default_filter_fact

Synopsis

Gets the default FilterFactory object.

OMG IDL

```
readonly attribute CosNotifyFilter::FilterFactory
        default_filter_factory;
```

Description

Used when subscribing. This operation is used in subscriber applications to get the default FilterFactory object.

Return Value

Default FilterFactory object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Getting the Event Channel, ConsumerAdmin Object, and Filter Factory Object.

C++ code example:

```
channel->default_filter_factory();
```

Java code example:

```
channel.default_filter_factory();
```

CosNotifyChannelAdmin::EventChannelFactory Class

This class is used by event poster applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface EventChannelFactory {
    EventChannel get_event_channel ( in ChannelID id )
    raises (ChannelNotFound);
  };
}; //CosNotifyChannelAdmin
```

CosNotifyChannelAdmin::EventChannelFactory::get_event

Synopsis

Gets the EventChannel object.

OMG IDL

```
EventChannel get_event_channel ( in ChannelID id )
   raises (ChannelNotFound);
```

Exceptions

CosNotifyChannelAdmin::ChannelNotFound

Indicates the channel cannot be found.

Description

Used when subscribing, unsubscribing, and posting events. This operation is used in applications to get the EventChannel object. When subscribing, the EventChannel object is used to get the filter factory object and the ConsumerAdmin object. When unsubscribing, the EventChannel object is used to get the ConsumerAdmin object.When posting an event, the EventChannel object is used to get the SupplierAdmin object. The ChannelID parameter that is passed in must be set to Tobj_Notification::DEFAULT_CHANNEL ; otherwise, the ChannelNotFound exception is raised.

Return Value

Returns the default event channel's object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Getting the Event Channel and Getting the Event Channel, ConsumerAdmin Object, and Filter Factory Object.

C++ code example:

```
channel_factory->get_event_channel(
   Tobj_Notification::DEFAULT_CHANNEL );
```

Java code example:

```
channel_factory.get_event_channel(DEFAULT_CHANNEL.value);
```

CosNotifyComm::StructuredPushConsumer Interface

This interface is used by event subscriber applications for event delivery. You must implement this interface so that the Notification Service can invoke on it to deliver events

to subscribers. It has three methods which you have to implement.

The OMG IDL for this class is as follows:

```
Module CosNotifyComm
{
 interface StructuredPushConsumer : NotifyPublish {

   void push_structured_event(
       in CosNotification::StructuredEvent event)
    raises(CosEventComm::Disconnected);
   void disconnect_structured_push_consumer:
   //The following operations are inherited.
   void offer_change(
       in CosNotification::EventTypeSeq added,
       in CosNotification::EventTypeSeq removed )
    raises ( InvalidEventType );
 };
}; //CosNotifyComm
```

CosNotifyComm::StructuredPushConsumer::push_structu

Synopsis

Delivers a structured event.

OMG IDL

```
void push_structured_event(
       in CosNotification::StructuredEvent event)
    raises(CosEventComm::Disconnected);
```

Exceptions

CosEventComm:: Disconnected

The subscriber should never raise this exception.

Description

Used when subscribing. This operation is implemented by the subscriber's callback object and is invoked by the Notification Service each time a structured event is delivered. This operation contains a single input parameter, which is a structured event.

Note: This operation will not be called in a transaction. Also, when this operation is called, it must return quickly because the Notification Service might not start delivering events to other subscribers until this operation returns.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see Implementing the CosNotifyComm::StructuredPushConsumer Interface.

C++ code example:

```
virtual void push_structured_event(
    const CosNotification::StructuredEvent& notification );
    {
    // Process the event.
    }
```

Java code example:

```
public void push_structured_event(StructuredEvent notification)
```

```
{
// Process the event.
}
```

CosNotifyComm::StructuredPushConsumer:: disconnect_structured_push_consumer

Synopsis

Never invoked.

OMG IDL

```
void disconnect_structured_push_consumer;
```

Description

This operations is never invoked. The subscriber application must provide a stubbed-out version of this operation.

Examples

C++ code example:

```
virtual void push_structured_event(
    const CosNotification::StructuredEvent& notification );
{
  throw new CORBA::NO_IMPLEMENT();
}
```

Java code example:

```
public void disconnect_structured_push_consumer()
{
  throw new CORBA::NO_IMPLEMENT();
}
```

CosNotifyComm::StructuredPushConsumer::Offer_change

Synopsis

Never invoked.

OMG IDL

```
void offer_change(
      in CosNotification::EventTypeSeq added,
      in CosNotification::EventTypeSeq removed )
   raises ( InvalidEventType );
```

Exceptions

CosNotifyComm::InvalidEventType

The subscriber should never raise this exception.

Description

This operations is never invoked. The subscriber application must provide a stubbed-out version of this operation.

Examples

C++ code example:

```
virtual void offer_change(
    const CosNotification::EventTypeSeq& added,
    const CosNotification::EventTypeSeq& removed )
  {
   throw CORBA::NO_IMPLEMENT();
  }
```

Java code example:

```
public void offer_change(EventType[] added, EventType[] removed)
{
 throw new NO_IMPLEMENT();
}
```

Exception Minor Codes

This section provides information about the Notification Service exception symbols and minor codes. The minor codes are in the Tobj_Events.idl and Tobj_Notification.idl files. These files are located in the wledir\include directory (for Microsoft Windows NT systems) and wledir/include directory (for UNIX systems).

Table 2-4 and Table 2-5 list the exception symbols and corresponding minor codes for the Tobj_Events and Tobj_Notification exceptions respectively. CORBA system events have a minor code field and those minor codes are also defined in these tables as well.

Note: The exception symbols are organized within the tables by the higher-level exceptions (CORBA::IMP_LIMIT , CORBA::CORBA::BAD_PARAM , CORBA::BAD_INV_ORDER , CORBA::INV_OBHJREF , and CORBA::OBJECT_NOT_EXIST) and listed in alphabetical order.

Table 2-4 Tobj_Events Exception Minor Codes

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
CORBA::IMP_LIMIT Exceptions		
Tobj_Events:: POST_DOMAIN_AND_TYPE_TOO_LONG This exception is raised by: • Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user specified a domain name and type name whose combined length was greater than 31 characters.	5455580D
Tobj_Events:: POST_DOMAIN_CONTAINS_SEPARATOR This exception is raised by: • Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user specified a domain name that contained the "." character.	54555802
Tobj_Events::POST_EMPTY_DOMAIN This exception is raised by: • Tobj_SimpleEvents::Channel::	When posting an event, the user specified an empty domain name.	5455580B

push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event		
Tobj_Events::POST_EMPTY_TYPE This exception is raised by: • Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user specified an empty type name.	5455580C
Tobj_Events:: POST_SYSTEM_EVENTS_UNSUPPORTED This exception is raised by: • Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user tried to post a TUXEDO system event; that is, the domain name is "TMEVT" and the type name starts with the "." character).	54555804
Tobj_Events:: POST_TYPE_CONTAINS_SEPARATOR This exception is raised by: • Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user specified a type name that contained the "." character.	54555803
Tobj_Events:: POST_UNSUPPORTED_PRIORITY_VALUE This is exception is raised by: • Tobj_SimpleEvents:: Channel::push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user added a "Priority" field in the variable header. However, the user did not set the field's value to a "short" in the range of 1 - 100.	54555801
Tobj_Events:: POST_UNSUPPORTED_VALUE_IN_ANY This exception is raised by: • Tobj_SimpleEvents:: Channel::push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user put an unsupported type (for example, a structure, union, sequence, etc.) into one of the "anys" in the structured event field. The unsupported type is in the variable header's value field, the filterable data's value field, or the remainder_of_body field.	54555800

Tobj_Events:: SUB_DOMAIN_AND_TYPE_TOO_LONG This exception is raised by: • Tobj_SimpleEvents::Channel:: subscribe • CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: connect_structured_push_consumer	When subscribing, the user specified a domain name and type name whose combined length is greater than 255 characters.	54555809
Tobj_Events:: SUB_DOMAIN_BEGINS_WITH_SYSEV This exception is raised by: • Tobj_SimpleEvents::Channel::subscribe • CosNotifyFilter::Filter::add_constraints	When subscribing, the user specified a domain name that begins with the "." character.	54555805
Tobj_Events::SUB_EMPTY_DOMAIN This exception is raised by: • Tobj_SimpleEvents::Channel:: subscribe • CosNotifyFilter::Filter:: add_constraints	The user specified an empty domain name when subscribing.	54555807
Tobj_Events::SUB_EMPTY_TYPE This exception is raised by: • Tobj_SimpleEvents::Channel:: subscribe • CosNotifyFilter::Filter:: add_constraints	The user specified an empty type name when subscribing.	54555808
Tobj_Events::SUB_FILTER_TOO_LONG This exception is raised by: • Tobj_SimpleEvents::Channel:: subscribe • CosNotifyFilter::Filter:: add_constraints	The user specified a data filter expression longer than 255 characters.	5455580A
Tobj_Events::SUB_NAME_TO_LONG • This exception is raised by: Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When subscribing, the user specified a subscription name longer than 127 characters.	5455580E
Tobj_Events:: TRANSIENT_ONLY_CONFIGURATION This exception is raised by:	The user tried to create a persistent subscription, but the system was configured to support transient subscriptions only.	54555806

• Tobj_SimpleEvents::Channel:: subscribe • CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: connect_structured_push_consumer		
CORBA::BAD_PARAM Exceptions		
Tobj_Events::INVALID_CHANNEL_ID This exception is raised by: • Tobj_SimpleEvents::ChannelFactory::find_channel	When looking up the channel using the Simple Events API, the user specified an invalid channel id, that is, a channel id that is not Tobj_SimpleEvents:: DEFAULT_CHANNEL .	54555813
Tobj_Events:: INVALID_SUBSCRIPTION_ID This exception is raised by: • Tobj_SimpleEvents::Channel:: unsubscribe • CosNotifyChannelAdmin:: ConsumerAdmin::get_proxy_supplier • Tobj_SimpleEvents:: Channel::exists	When unsubscribing using the Simple Events API, the user specified an invalid subscription id, that is, a non-existent or a CosNotification subscription id. When looking up a subscription using the CosNotification Service API, the user specified an invalid subscription id, that is, a non-existent or a Simple Events API subscription id. When calling the exists operation using the BEA Simple Events API, the user passed in a CosNotification subscription_id .	54555812
Tobj_Events:: SUB_INVALID_FILTER_EXPRESSION This exception is raised by: • Tobj_SimpleEvents::Channel:: subscribe • CosNotifyFilter::Filter:: add_constraints	When subscribing, the user specified an invalid data filter expression. This either means that there is a syntax error in the expression or that one of the field names in the expression is not defined as an FML field. Check that you have correctly created FML field tables that contain all fields that you want to data filter on, and check that the UBBCONFIG file is properly configured so that the field table files can be found.	54555810
Tobj_Events:: SUB_UNSUPPORTED_QOS_VALUE This exception is raised by: • Tobj_SimpleEvents::Channel:: subscribe • CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: set_qos	**Minor Code: 54555811** When subscribing, the user specified an invalid subscription quality of service. For the Simple Events API, this means that the quality of service specified did not meet one of the following requirements: • The sequence must be of length one. • The name must be Tobj_SimpleEvents:: SUBSCRIPTION_TYPE . • The value must be either Tobj_SimpleEvents:: TRANSIENT_SUBSCRIPTION or Tobj_SimpleEvents:: PERSISTENT_SUBSCRIPTION For the CosNotification Service API, this means that the quality of service specified did not meet one of the following requirements: • The quality of service must contain a name/value pair where the name is Tobj_Notification:: SUBSCRIPTION_TYPE and the value is Tobj_Notification:: TRANSIENT_SUBSCRIPTION or Tobj_Notification:: PERSISTENT_SUBSCRIPTION	

	• The quality of service may contain a name/value pair where the name is Tobj_Notification::SUBSCRIPTION_NAME and the value is a string containing the subscription's administrative name.	
CORBA::INV_OBHJREF		
Tobj_Events:: SUB_NIL_CALLBACK_REF This exception is raised by: • Tobj_SimpleEvents::Channel:: subscribe • CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: connect_structured_push_consumer	When subscribing, the user specified a NIL object reference for the callback object which receives events.	54555830

Table 2-5 Tobj_Notification Exception Minor Codes

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
CORBA::IMP_LIMIT Exceptions		
Tobj_Notification:: SUB_ADD_CONS_ON_TIMED_OUT_FILTER This exception is raised by: CosNotifyFilter::Filter:: add_constraints	A CosNotification subscriber waited more than 5 minutes after creating a filter to call add_constraints on the filter. This means that the filter has been destroyed (timed out) and the subscriber must create a new filter.	54555858
Tobj_Notification:: SUB_ADD_CONS_TO_ADDED_FILTER This exception is raised by: CosNotifyFilter::Filter:: add_constraints	A CosNotification subscriber called add_constraints on a filter that had already been added to a proxy.	5455585E
Tobj_Notification:: SUB_ADDED_TIMED_OUT_FILTER This exception is raised by: CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: add_filter	After creating a filter and calling "add_constraints" on it, a CosNotification subscriber waited more than 5 minutes to call add_filter to add the filter to the proxy. This means that the filter has been destroyed (timed out) and that the subscriber must create a new filter.	5455585D
Tobj_Notification:: SUB_ADD_FILTER_AFTER_CONNECT CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: add_filter	A CosNotification subscriber called add_filter after connecting to the proxy.	54555852
Tobj_Notification:: SUB_CANT_SET_QOS_AFTER_CONNECT This exception is raised by: CosNotifyChannelAdmin::StructuredProxyPushSupplier::set_qos	A CosNotification subscriber called set_qos after connecting to the proxy.	54555856
Tobj_Notification:: SUB_MULTIPLE_CALLS_TO_ADD_CONS This exception is raised by: CosNotifyFilter::Filter:: add_constraints	A COS subscriber called add_constraints more than once on a filter.	54555859

Tobj_Notification:: SUB_MULTIPLE_CALLS_TO_SET_FILTER This exception is raised by: CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: add_filter	A CosNotification subscriber called add_filter more than once on a proxy.	54555851
Tobj_Notification:: SUB_MULTIPLE_CALLS_TO_SET_QOS This exception is raised by: CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: set_qos	A CosNotification subscriber called set_qos more than once on a proxy.	54555855
Tobj_Notification:: SUB_MULTIPLE_CONSTRAINTS_IN_LIST This exception is raised by: CosNotifyFilter::Filter:: add_constraints	When a CosNotification subscriber called add_constraints on a filter, the subscriber passed in a list of constraints that had more than one item; that is, the subscriber was trying to send in a list of data filters instead of one data filter.	5455585A
Tobj_Notification:: SUB_MULTIPLE_TYPES_IN_CONSTRAINT This exception is raised by: CosNotifyFilter::Filter:: add_constraints	When a COS subscriber called add_constraints on a filter, the subscriber passed on a constraint that had more than one domain/type set; that is, the subscriber was trying to send in a list of desired event types instead of one event type.	5455585B
Tobj_Notification:: SUB_NIL_FILTER_REF This exception is raised by: CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: add_filter	A CosNotification subscriber passed a nil filter object reference into add_filter .	54555853
Tobj_Notification:: SUB_NO_CUSTOM_FILTERS This exception is raised by: CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: add_filter	A CosNotification subscriber passed a filter object that was not created by the default filter factory into add_filter . For example, a CosNotification subscriber implemented the CosNotifyFilter::Filter interface to do some kind of "custom" filtering and passed one of those filter objects into add_filter .	54555854
Tobj_Notification:: SUB_SET_FILTER_NOT_CALLED This exception is raised by: CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: connect_structured_push_consumer	A CosNotification subscriber did not call add_filter to the proxy before connecting to the proxy.	54555850
Tobj_Notification:: SUB_SET_QOS_NOT_CALLED This exception is raised by: CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: connect_structured_push_consumer	A CosNotification subscriber did not call add_filter to the proxy before connecting to the proxy.	54555857
Tobj_Notification:: SUB_SYSTEM_EVENTS_UNSUPPORTED This exception is raised by: CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: set_qos	A CosNotification subscriber passed in a domain name of "TMEVT" and a type name that begins with "." ; that is, the CosNotification subscriber was trying to subscribe to TUXEDO system events. This is not supported. It is only supported by the Simple Events API.	5455585C

Tobj_Notification:: SUB_UNSUPPORTED_CLIENT_TYPE This is raised by: • ConsumerAdmin:: obtain_notification_push_ supplier • SupplierAdmin:: obtain_notification_push_ consumer	When creating a proxy, a CosNotification subscriber or poster passed in a client type other than CosNotifyChannelAdmin::STRUCTURED_EVENT .	5455585F
CORBA::OBJECT_NOT_EXIST Exception		
Tobj_Notification:: SUBSCRIPTION_DOESNT_EXIST This exception is raised by: • StructuredProxyPushSupplier:: add_filter • StructuredProxyPushSupplier:: set_qos • StructuredProxyPushSupplier:: connect_structured_push_ consumer • StructuredProxyPushSupplier:: disconnect_structured_push_ supplier **Note:** connect_structured_push_ consumer can raise this exception since a user can create the proxy, then use the ntsadmin utility to delete the subscription, and then call connect_structured_push_ consumer on the proxy.	A CosNotification subscriber called a method on a proxy that had already been destroyed. The proxy has been destroyed by one of the following actions: • The CosNotification subscriber disconnected the proxy. • The CosNotification subscriber waited more than five minutes from creating the proxy to connecting it; that is, it took longer than five minutes to complete the subscription. • The administrator used the ntsadmin utility to destroy the subscription.	54555880

BACK TO TOP ▲